1 Wozu das Ganze?

Landkarten im Stile von Google-Maps u.ä. gibt es ja reichlich, aber eigentlich alle Projekte arbeiten mit Bildformaten, die sich zwar auf dem PC recht gut handhaben lassen, aber in einem embedded System denn doch größere Schwierigkeiten bereiten.

Dieses Projekt ist ein Versuch, Landkarten auch im embedded Bereich darstellen zu können, wo es eher schwerfällt, JPG oder PNG zu dekodieren und wo man es auch eher schwer hat, mit einer großen Anzahl kleinerer Dateien für die einzelnen Tiles umzugehen.

Ich habe mir deswegen ein paar Gedanken gemacht und heraus gekommen ist eine Lösung, die man auch auf einem eher mäßig ausgestatteten ARM Board realisieren kann.

Das Prinzip ist so:

Erster Schritt: Man stellt sich mit einem geeigneten Programm, z.B. dem Mapgenerator für Trekbuddy, seine Landkarte im sogenannten "tared trekbuddy" Format zusammen. Welche Zoom-Levels man haben will, muß man selber entscheiden. (Trekbuddy ist eine JAVA-ME Anwendung)

Zweiter Schritt: mit dem beigefügten Programm MapConvert.exe konvertiert man jeden Zoom-Level (einzeln) in mein Kartenformat. Im Prinzip Tar-File rein, Level-File raus.

Diese Level-Files kopiert man dann auf seine SD-Karte, die man dann mit seinem uC-Board verwendet. Ein Tip noch für all die Atmel-Fans: Landkartenprojekte sind für einen ATmega ein paar Nummern zu groß - also bitte nicht herumklagen, daß man Berlin im Zoomlevel 16 (130 MB) nicht mehr gehandelt kriegt.

Das Grundprinzip bleibt gleich: Es besteht die Karte aus Grafik in quadratischen Stücken (Tiles) zu je 256x256 Pixeln, die zur Ansicht passend nebeneinander bzw. untereinander auf einem Farb-LCD dargestellt werden. Der Fortschritt (aus Sicht des uC-Programmierers) besteht darin, daß die Bitmaps der Tiles nicht mehr aus PNG oder JPG Grafiken bestehen, sondern in einem Format gespeichert sind, das sich mit einem sehr einfachen und (auf 32 Bit-Systemen) schnellen Algorithmus dekomprimieren läßt.

2 Die Tiles

Der Ansatz zur Lösung ist zunächst, daß man alle Tiles (256x256 Pixel große Kacheln) eines Levels (Vergrößerungsstufe) in einer größeren Datei zusammenzufassen. Simples BMP wäre natürlich zwar das Einfachste, ist aber doch recht platzverschwenderisch. Deshalb ist schon eine gewisse Kompression anzustreben, die aber einfachst dekodierbar sein muß, denn so viel Rechenleistung wie am PC hat man in einem kleinen uC eben nicht.

Wie also kommen wir nun zu einer einfachen Kompression? Nun, Landkarten sind was anderes als Fotos von der Liebsten. Sie kommen mit sehr viel weniger Farben aus als echte Bilder und so können wir die Anzahl der Farben durchaus saftig reduzieren, ohne daß darunter die Darstellung leidet. Für weniger Farben braucht man auch weniger Bits.

Gewählt habe ich zur Darstellung 48 Farben per Palette. Die lassen sich in 6 Bit (0..63) darstellen und es bleiben noch 16 Codes übrig (48..63), die dazu verwendet werden, mehrere gleiche aufeinanderfolgende Pixel zu codieren. Das ist zwar keine hocheffektive Kompression, aber es reicht für unsere Zwecke aus.

Jedes Tile besteht aus einem Header, daran folgend ein Satz Paletteneinträge und daran folgend die Pixel, nach obigem Schema komprimiert. Die enthaltene Palette ist nur 16 Bit breit. Sie liefert deshalb nur 64K Farben im 454-Schema: 4 Bit rot, 5 Bit grün, 4 Bit blau. Für

Embedded reicht das völlig, aber für das PC-Demo-Programm, das mit 24 Bit Farbe arbeitet, hab ich das auf 24 Bit aufgebläht, um mir das Umformatieren der Windows-Bitmaps zu sparen.

Hier nun etwas Quelltext:

```
  { der Kachelheader fuer mein 6 Bit Kachel-Format }
  PHeader = ^THeader;
  THeader = packed record
   magic  : array[0..2] of char;         // 'BQM'
   zoom   : byte;       { Zoom-Stufe }
   dx     : smallword;  { little endian, also }
   dy     : smallword;  { zuerst kommt lo, dann hi }
   Top    : longint;    { LAT von oberer  Kante * 1000000 }
   Bot    : longint;    { LAT von unterer Kante * 1000000 }
   Left   : longint;    { LON von linker  Kante * 1000000 }
   Right  : longint;    { LON von rechter Kante * 1000000 }
   PalLen : smallword;  { phys. Laenge = 2 * PalLen }
   end;
```

Hier nun die Dekodierung in Pascal geschrieben:

```
var
  MapBuf    : PBuffer;   // Zeiger auf den Bereich der aktuellen Kachel
  Palette   : array[0..47] of longint; // hier für Windows 24 Bit Color
  PixPos    : longint;
  BitPos    : longint;
  BitBuf    : longint;
  qOld,
  qRep      : byte;
  TLH       : THeader;
  i, j      : integer;
  w         : word;


begin
  // zunächst die Palette füllen
  L:= TLH.PalLen;
  FillChar(Palette,sizeof(Palette),0);
  if L > 47 then L:= 47;
  j:= sizeof(THeader);
  for i:= 0 to L do
  begin
    w:= MapBuf^[j+1];            // byteweise, weil packed
    w:= (w shl 8) or MapBuf^[j]; // und deshalb unaligned
    inc(j,2);
    // Verrenkung für 24 Bit Colors
    r := (w shl 3) and $F8;
    g := (w shr 3) and $FC;
    b := (w shr 8) and $F8;
    Palette[i]:= (r shl 16) or (g shl 8) or b;
  end;


  // Position des 1. Pixels in dem aktuellen Tile-Block nach PixPos laden
  PixPos:= (aTile.TLH.PalLen * 2) + sizeof(THeader);

  // Bit-Dekoder initialisieren
  BitPos:= 0;
  BitBuf:= 0;
  qRep:= 0;
  qOld:= 0;
```

```
  // alle Pixel in einen Puffer laden
  for i:= 0 to 255 do
  for j:= 0 to 255 do
  begin
    L:= GetPixel;
    Puffer[j, i]:= L;
  end;
end
```

Dazu braucht es dann noch die entsprechenden Routinen:

```
function PxGet: byte;
var
  Q : longint;
begin
  if BitPos<6
  then begin
         Q:= MapBuf^[PixPos];
         inc(PixPos);
         BitBuf:= BitBuf or (Q shl BitPos);
         inc(BitPos,8);
       end;
  result:= BitBuf and $3F;
  BitBuf:= BitBuf shr 6;
  dec(BitPos,6);
end;



function GetPixel: longint;
var
  b: byte;
begin
  if qRep>47
  then
  begin
    result:= Palette[qOld];
    dec(qRep);
  end
  else
  begin
    b:= PxGet;
    if b<48
    then begin
           qOld:= b;
           result:= Palette[b];
         end
    else begin
           result:= Palette[qOld];
           qRep:= b-1;
         end;
  end;
end;
```

Das ist der ganze Dekodier-Algorithmus. Ist sehr leicht nach C oder sonstwas umzuschreiben. Als Input braucht es das Tile (der? die? das? Tile) in einem Pufferbereich (array of bytes), wo MapBuf drauf zeigt und als Output hab ich hier mal einfach einen Puffer[j, i] geschrieben. In einem uC-System würde man hier sicherlich mehr oder weniger direkt den Bildspeicher adressieren und dabei das nötige Clipping usw. vornehmen, damit man nicht danebenschreibt.

3 Die Levels

Alle diese Tiles sollte man natürlich nicht einzeln auf eine SD-Karte o.ä. speichern, denn das wäre die blanke Platzvergeudung. Also kommen sie in ein Levelfile, das so strukturiert ist:

1. ein Header

2. eine Offset-Tafel, deren Einträge auf die einzelnen Tiles zeigen

3. Die Tiles selbst.

Ich habe es hier erstmal so gehandhabt, daß jedes Tile bei einer glatten Sektoradresse (also n mal 512) beginnt. Vielleicht ist dadurch der Aufwand im Fat-Filesystem nicht so groß, wer weiß.. (allerdings bläht es das Levelfile auf)

Hier nun das Format des Headers:

```
{ der Dateiheader für mein eigenes Map-Format }
PLevelHeader = ^TLevelHeader;
TLevelHeader = packed record
  Magic  : array[0..3] of char;        // 'QMFT'
  numX   : longint;   { Anzahl Kacheln in X-Richtung }
  numY   : longint;   { Anzahl Kacheln in Y-Richtung }
  Top    : longint;    { LAT von oberer  Kante * 1000000 }
  Bot    : longint;    { LAT von unterer Kante * 1000000 }
  Left   : longint;    { LON von linker  Kante * 1000000 }
  Right  : longint;    { LON von rechter Kante * 1000000 }
  Zoom   : byte;       { Zoomstufe 0..19 oder so}
  Name   : array[0..98] of char; { Kartenname usw. }
  end;
```

und direkt daran die Offset-Tafel:

```
Offsets: array[0..(numX*numY)-1] of longint;  // die Positionen der
                                              // Kacheln in der Datei
```

und dann die Tiles:

```
Kachel[0];
Kachel[1];
....
Kachel[numX*numY - 1];
ende.
```

Adressiert werden die Tiles so:

```
long L, M;
 M = WantedTileNumber_X + (WantedTileNumber_Y * (LevelHeader->numX));
 L = Offsets[M];
 Seek(MapFile,L);
```

4 Koordinaten

Sowohl im Tile-Header als auch im Level-Header habe ich es vermieden, mit double zu arbeiten, denn die jeweiligen Implementationen können voneinander abweichen und das Gleitkommarechnen sollte man in embedded Systemen nur dort tun, wo es wirklich nötig ist.

Statt double wird long verwendet, und zwar Längen- und Breitengrade mal 1 Million. Mit Long kann man +/- 2000 Millionen darstellen, es ist also genug Platz da für 0..360 Grad bzw. -128..+128 Grad.

5 Das Kodierprogramm

Anbei gibt es auch ein Programm, um solche Levelfiles zu erzeugen. Ich habe mir nach einigem Testen das "tared Trekbuddy"-Format ausgesucht. Trekkingleute kennen das und es gibt dafür auch passable Programme, um aus Quellen wie OpenStreetmap udgl. derartige für das Programm Trekbuddy geeignete Karten zu erzeugen. Dieses Format besteht aus einer gewissen formalen Verzeichnisstruktur, die auch bei meinem Programm benötigt wird:

```
[kartenverzeichnis]                     das generelle verzeichnis
          ["mapname" nn]                subdir für Zoomlevel nn
          ["mapname" mm]                subdir für Zoomlevel mm
          ["mapname" kk]                subdir für Zoomlevel kk
          ["mapname" xx (0)]            subdir 0 für Zoomlevel xx
          ["mapname" xx (1)]            subdir 1 für Zoomlevel xx
          ["mapname" xx (2)]            subdir 2 für Zoomlevel xx
          ["mapname" xx (3)]            subdir 3 für Zoomlevel xx
```

Bei Leveln, deren Pixelanzahl in X und/oder Y Richtung größer ist als 32511, wird die Karte in mehrere Stücke geteilt. Deshalb hat es die 4 Verzeichnisse für Level xx.

In jedem Level-Verzeichnis finden sich dann 2 Dateien:

```
"mapname" nn.tar
"mapname" nn.tmi
```

bzw.

```
"mapname" nn (0).tar              bzw. 1,2,3 bei geteilten Mapleveln
"mapname" nn (0).tmi
```

Die .tar enthält die Tiles und die Koordinaten und die .tmi ist eine Indexdatei, die die Positionen der Tiles innerhalb der .tar enthält.

Um einen Level umzukodieren, wählt man zunächst ein Verzeichnis aus, wo die Ergebnisse gespeichert werden sollen. Dann wählt man das Verzeichnis aus, das die *.tar und *.tmi enthält. Man kann einen Kommentar eigener Wahl in die Zieldatei einbauen lassen. Mit dem Drücken des Buttons "Levelfile erzeugen" startet man das Umcodieren und darf ne Weile warten... Ich hätte den Konverter in einem separaten Thread laufen lassen können, bin aber bislang dazu zu faul gewesen. So ist die Oberfläche halt solange blockiert, bis das ganze fertig ist.

Nochwas zur Kompression:

So ganz schlecht sieht mein Format garnicht aus:

Europa Level 5:	TAR 380K	meins 340K
dito Level 6:	TAR 956K	meins 892K
dito Level 7:	TAR 3335K	meins 3419K
dito Level 8:	TAR 9845K	meins 9859K
dito Level 9:	TAR 40221K	meins 42747K

Mit höherem Zoom-Level kommt meine Kompression so langsam an ihre Grenzen. Man könnte jetzt was Ausgeklügelteres basteln, aber das macht nur das Dekomprimieren komplizierter und damit langsamer.

6 Nachtrag: Die Zeichenroutine (Delphi) des PC-Programmes:

(auf nem uC sieht das alles ganz anders aus, da hat man zumeist keine Windows-typischen Strukturen und Methoden - es sei denn, man benutzt WinCe)

```
procedure TMapCanvas.Paint(var Msg:TWMPaint);
var
  ps      : TPaintStruct;
  R       : TRect;
  hBmp,
  TmpDC,
  MemDC   : Integer;
  CurTile       : HBITMAP;
  Xmem, Ymem    : longint;     // LinksOben vom Mem-BMP
  Wmem, Hmem    : longint;     // Breite, Höhe vom Mem-BMP
  kx, ky        : longint;     // Kachelnummern
  dx, dy        : longint;     // Versatz Canvas zum BMP im MemDC
  nx, ny        : longint;     // Anzahl Kacheln in MemDC
  cx, cy        : longint;     // lfd. Kachel Nummer

begin
  BeginPaint(Handle,ps);
  if(Width>0)and(Height>0)
  then
  begin
    memDC:=CreateCompatibleDC(ps.hdc);
    TmpDC:=CreateCompatibleDC(ps.hdc);

    { die nächste gerade Pixeladresse vor XBild und YBild ermitteln }
    Xmem:= Xbild and $7FFFF00;
    Ymem:= Ybild and $7FFFF00;

    { die Breite und Höhe des zu erzeugenden Memory-Bitmaps errechnen }
    Wmem:= ((Xbild + Width + 255) and $7FFFF00) - Xmem;
    Hmem:= ((Ybild + Height + 255) and $7FFFF00) - Ymem;

    { ein Bitmap erzeugen, das auf allen Seite gleich oder
      größer ist als das anzuzeigende Bild. Offset dann in dx und dy }
    hBmp:=CreateCompatibleBitmap(ps.hdc,Wmem,Hmem);
    dx:= Xbild - Xmem;
    dy:= Ybild - Ymem;
    SelectObject(memDC,hBmp);

{ ------ alle benötigten Kacheln ermitteln und in den MemDC malen ------ }
    nx:= Wmem shr 8; if nx > 16 then nx:= 16;
    ny:= Hmem shr 8; if ny > 16 then ny:= 16;

    ky:= Ymem shr 8;
    cy:= 0;
    while cy <= ny do
    begin
      kx:= Xmem shr 8;
      cx:= 0;
      while cx <= nx do
      begin
        TagTile(kx,ky); // Cache aktualisieren
        inc(kx);
      inc(cx);
      end;
      inc(ky);
      inc(cy);
    end;
```

```
    ky:= Ymem shr 8;
    cy:= 0;
    while cy <= ny do
    begin
      kx:= Xmem shr 8;
      cx:= 0;
      while cx <= nx do
      begin
        if (kx<0) or (ky<0)
        then
        begin                    // wo kein Tile vorhanden --> grau machen
          R.Left:= cx*256;
          R.Top := cy*256;
          R.Right:= R.Left + 256;
          R.Bottom:= R.Top + 256;
          FillRect(MemDC, R, GetStockObject(LTGRAY_BRUSH));
        end
        else
        begin
        CurTile:= GetTile(kx,ky,fMouseMoveState);    // Tile laden
        if CurTile<>0
        then begin
               SelectObject(TmpDC, CurTile);
               BitBlt(MemDC,
               cx*256,cy*256,
               256,256,
               TmpDC,
               0,0,
               SRCCOPY);
             end
        else begin               // wo kein Tile vorhanden --> grau machen
               R.Left:= cx*256;
               R.Top := cy*256;
               R.Right:= R.Left + 256;
               R.Bottom:= R.Top + 256;
               FillRect(MemDC, R, GetStockObject(LTGRAY_BRUSH));
             end;
        end;
        inc(kx);
      inc(cx);
      end;
      inc(ky);
      inc(cy);
    end;

{ ------------------------------------------------------------------- }
    SelectObject(TmpDC,0);
    DeleteDC(TmpDC);

    BitBlt(ps.hdc,0,0,Width,Height,memDC,dx,dy,SRCCOPY);
    SelectObject(MemDC,0);
    DeleteDC(MemDC);
    DeleteObject(hBmp);
  end;
  if(csDesigning in ComponentState)
  then
  begin
    FillRect(ps.hdc,ps.rcPaint,Brush.Handle);
    DrawFocusRect(ps.hdc,RECT(0,0,Width,Height));
  end;
  EndPaint(Handle,ps);
end;
```